

# Assignment 11.2

## Task Description -1 – (Stack Using AI Guidance)

- Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

- A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows files task1.py, task2.py, task3.py, task4.py, and task5.py.
- Editor:** The active file is task1.py, which contains code for a simple LIFO stack implementation using a Python list. It includes methods for push, pop, peek, and is\_empty, along with example usage at the bottom.
- Terminal:** The terminal shows the execution of the code, demonstrating push, pop, peek, and is\_empty operations.
- Chat Panel:** The right-hand panel is titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON". It contains a tip about using Plan mode, requirements (Create a Stack class, Implement push(), pop(), peek(), and is\_empty() methods, Add proper comments explaining each method, Add example usage with sample output, Also explain time complexity of each operation), and a preview of the generated lab1.py file.
- Status Bar:** Shows the current line (Ln 10, Col 64), tab size (Tab Size: 4), encoding (UTF-8), file type (Python), version (3.13.1), Python version (Python 3.13 (64-bit)), and various tool icons.

Implemented a Stack class backed by a Python list.

Methods:

- `push(item)`: adds an element to the top.
- `pop()`: removes and returns the top element, raises `IndexError` if empty.
- `peek()`: returns (without removing) the top element, raises `IndexError` if empty.
- `is_empty()`: returns True if there are no elements.

## Task Description -2 – (Queue Design)

- Task: Use AI assistance to create a Queue data structure following FIFO principles

### Expected Output:

- A complete Queue implementation including enqueue, dequeue, front element access, and size calculation

```
File Edit Selection View Go Run Terminal Help < > Devops task1.py task2.py task3.py task4.py task5.py ... CHAT AI-ASSISTED CODING > task2.py ... DESIGNING A STACK DATA STRUCTURE IN PYTHON Requirements: Create a queue data structure in Python following FIFO principles. Implementing queue in task2.py I'll open your task2.py and then implement a commented Queue class with example usage and time complexity notes. Reviewed task2.py Now I'll write the Queue implementation, comments, explanation, and example usage into task2.py. Updated task2.py Here's your Queue implementation and how it satisfies each requirement: Implemented a Queue class in task2.py using collections.deque for efficient operations. Methods: enqueue(item): adds an element to the rear of the queue (FIFO). task5.py + task2.py GPT-5.1
```

task2.py content:

```
"""Queue implementation in Python following FIFO (First-In, First-Out) principle.

This module defines a Queue class with the following operations:
- enqueue(item)
- dequeue()
- front()
- size()

It also includes example usage at the bottom.

"""

from collections import deque

class Queue:
    """A simple FIFO (First-In, First-Out) queue data structure.

    Internally this uses collections.deque for efficient O(1) operations
    at both ends.
    """

    def __init__(self):
        """Initialize an empty queue.

        Enqueueing elements: 'A', 'B', 'C'
        Current queue: Queue(['A', 'B', 'C'])

        Front element: A

        Dequeueing elements (FIFO order):
        Dequeued: A
        Dequeued: B
        Current queue: Queue(['C'])

        Queue size: 1
    """

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items.pop(0)

    def front(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items[0]

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0
```

Terminal output:

```
C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task2.py"
Is queue empty? True
Enqueueing elements: 'A', 'B', 'C'
Current queue: Queue(['A', 'B', 'C'])

Front element: A

Dequeueing elements (FIFO order):
Dequeued: A
Dequeued: B
Current queue: Queue(['C'])

Queue size: 1
```

- Implemented a Queue class using collections.deque for efficient operations.
- Methods:
  - enqueue(item): adds an element to the rear of the queue (FIFO).
  - dequeue(): removes and returns the front element; raises IndexError if empty.
  - front(): returns (without removing) the front element; raises IndexError if empty.
  - size(): returns the current number of elements.
  - is\_empty(): helper to check if the queue has no elements.

## **Task Description -3 – (Singly Linked List Construction)**

- Task: Utilize AI to build a singly linked list supporting insertion and traversal.

## **Expected Output:**

- Correctly functioning linked list with node creation, insertion logic, and display functionality.

- Added Node and LinkedList classes.
  - Node holds data and a next reference that points to the next node (or None for the last node).
  - LinkedList:
    - Maintains head (first node).
    - `insert_at_end(data)`: creates a new node and links it at the end by walking from head to the last node and updating its next.
    - `traverse()`: walks from head via next, collects data values into a Python list, and returns it.
    - `display()`: prints the list in the form 10 -> 20 -> 30 -> None.

## Task Description -4 – (Binary Search Tree Operations)

- Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.

Expected Output:

- BST program with correct node insertion and in-order traversal output.

The screenshot shows the VS Code interface with several tabs open. The main tab is `task4.py`, which contains the following code:

```
AI-ASSISTED CODING > task4.py > ...
1  """Binary Search Tree (BST) implementation in Python.
2
3 This file defines:
4 - Node class: represents a single node in the BST
5 - BinarySearchTree class: supports insertion and in-order traversal
6
7 Features Implemented:
8 - Insert a value into the BST
9 - In-order traversal (Left, Root, Right)
10
11 Example usage is provided at the bottom.
...
14
15 class Node:
16     """A single node in a Binary Search Tree.
17
18     Each node stores:
19         - data: the value stored at this node
20         - left: reference to the left child (values < data)
21         - right: reference to the right child (values > data)
22     """
23
24     def __init__(self, data):
25         self.data = data
26         self.left = None
27         self.right = None
28
29
30 class BinarySearchTree:
31     """Binary Search Tree (BST) with recursive insertion and in-order traversal."""
32
33     def __init__(self):
34         # Root of the BST. Initially, the tree is empty so root is None.
35         self.root = None
36
37     def insert(self, value):
38         """Insert a value into the BST.
...
(c) Microsoft Corporation. All rights reserved.
```

The terminal tab shows the command `python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task4.py"` and the output of inserting values into the BST: [50, 30, 70, 20, 40, 60, 80]. It also shows the in-order traversal result: [20, 30, 40, 50, 60, 70, 80].

The status bar indicates the file is 3.13.1, Python 3.13 (64-bit), and the code is in AI-ASSISTED\_CODING/task4.py.

Added Node and BinarySearchTree classes in AI-ASSISTED\_CODING/task4.py.

Node holds data, left, and right references.

BinarySearchTree:

`insert(value)`: public insert method using `_insert_recursive(node, value)`:

Base case: if node is None, create and return a new Node.

Recursive case: go left if `value < node.data`, right if `value > node.data`, then return node to maintain links.

`inorder_traversal()`: returns a list of values; uses `_inorder_recursive(node, result)`:

Base case: node is None → return.

Recursive case: traverse left, visit node (append data), traverse right.

Comments in both recursive helpers explain base/recursive cases clearly.

## **Task Description -5 – (Hash Table Implementation)**

- Task: Create a hash table using AI with collision handling

## **Expected Output:**

- Hash table supporting insert, search, and delete using chaining or open

```

Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]
Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task5.py"
Inserting key-value pairs:
Hash table internal state (buckets):
HashTable([[(‘apple’, 1), (10, ‘ten’), (15, ‘fifteen’)], [], [], [(‘grape’, 3)], [(‘banana’, 2)]))

Searching for keys:
search(‘apple’) -> 1
search(‘banana’) -> 2
search(‘grape’) -> 3
search(10) -> ten
search(15) -> fifteen
search(‘unknown’) -> None

Deleting keys:
delete(‘banana’) -> True
delete(10) -> True
delete(‘unknown’) -> False

```

- HashTable class with:
  - `__init__(capacity=10)`: creates capacity buckets, each an empty list.
  - `_hash(key)`: uses Python `hash(key) % capacity` to pick a bucket index.
  - `insert(key, value)`: updates existing key or appends `(key, value)` into the bucket.
  - `search(key)`: scans the bucket for key, returns the value or `None`.
  - `delete(key)`: removes `(key, value)` from the bucket, returns `True/False`.
- Chaining:
  - `self.table` is a list of lists (buckets).
  - Each bucket stores multiple `(key, value)` pairs that share the same index → this is collision handling by chaining.